Figure 1
Modified SSA-conversion process

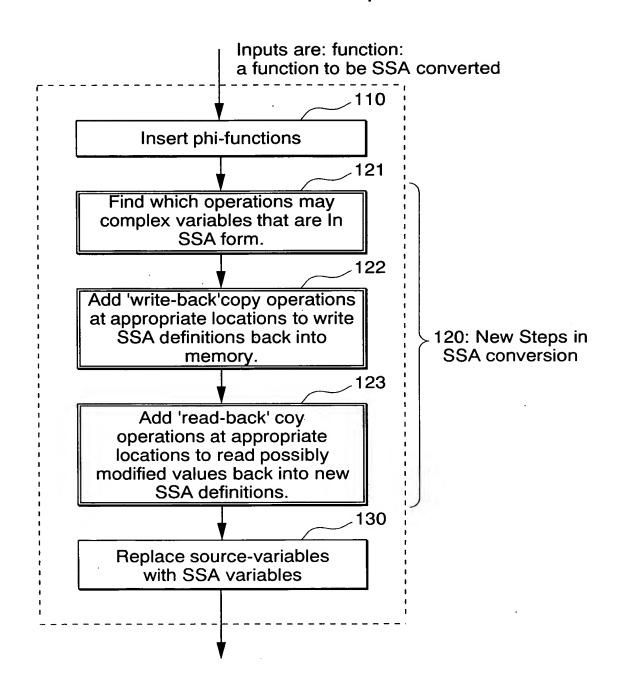


Figure 2
Overall compiler control flow

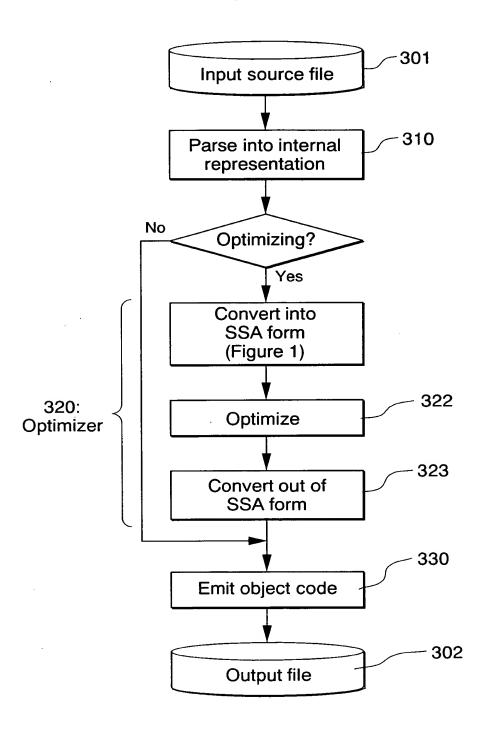


Figure 3
Program representation

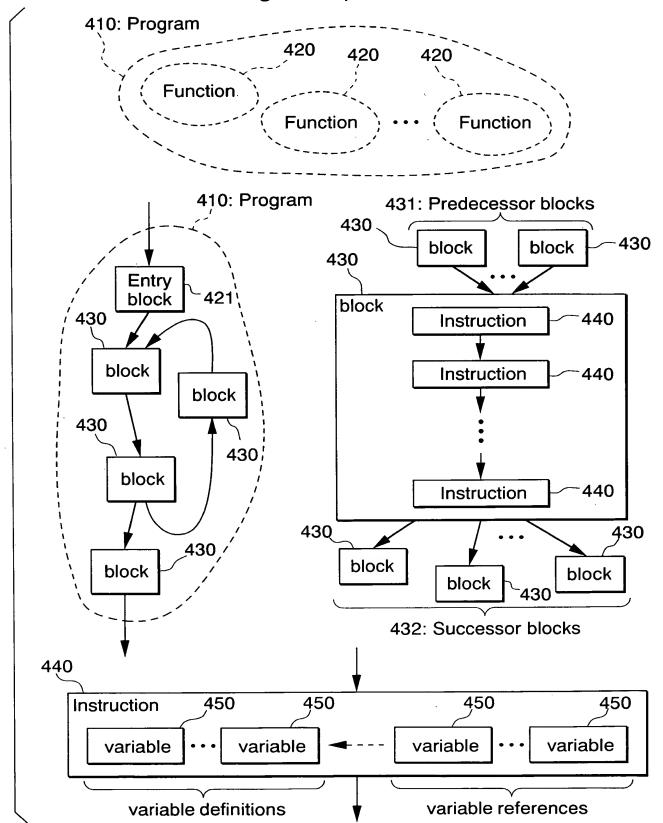
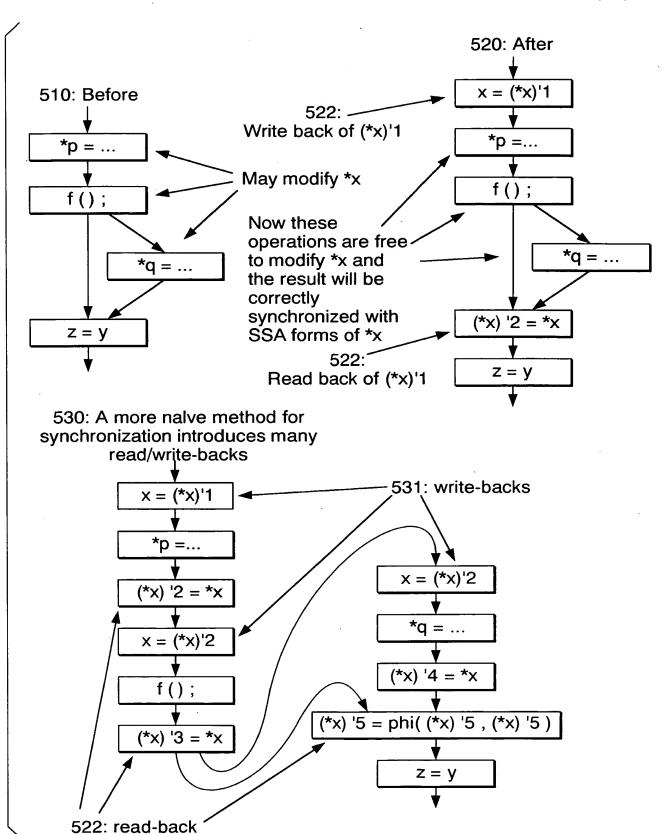
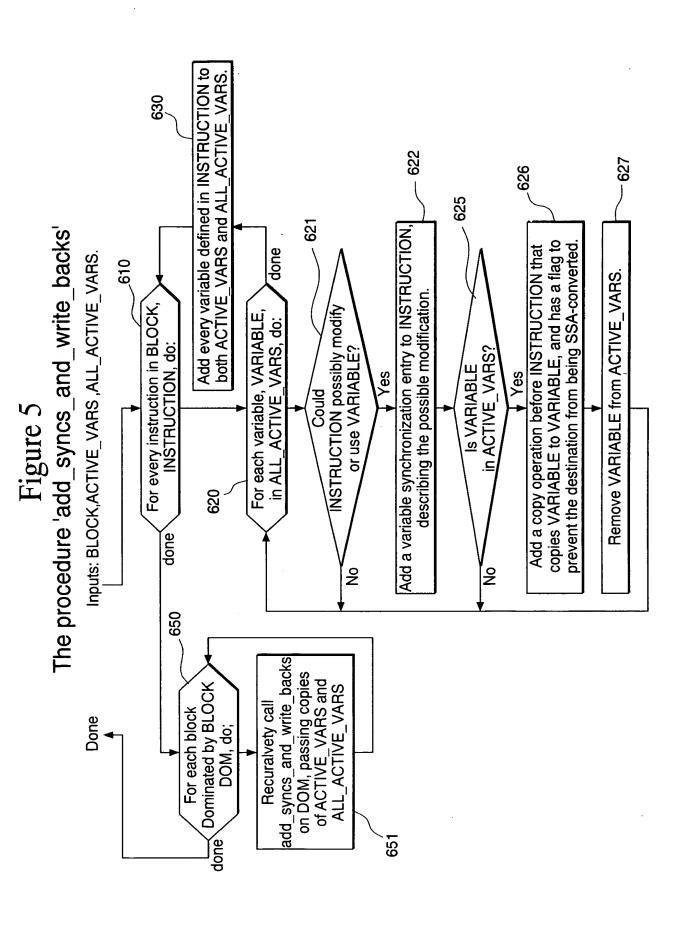


Figure 4
Placement of read/write-backs for the SSA form of *x, (*x)'1





70 - 702 710 Conversion step (a'.III), insertion of read-backs Invoke the procedure 'propagate_block_read_backs' on Initialize the mappings BLOCK_BEGIN_READ_BACK and BLOCK_END_READ_BACKS to empty mappings. While PENDING_BLOCKS is not empty, do: Initialize the queue PENDING_BLOCKS to contain only the function's entry block Inputs: a function Figure 6 done Done

711

the head of the queue, after removing its from the queue

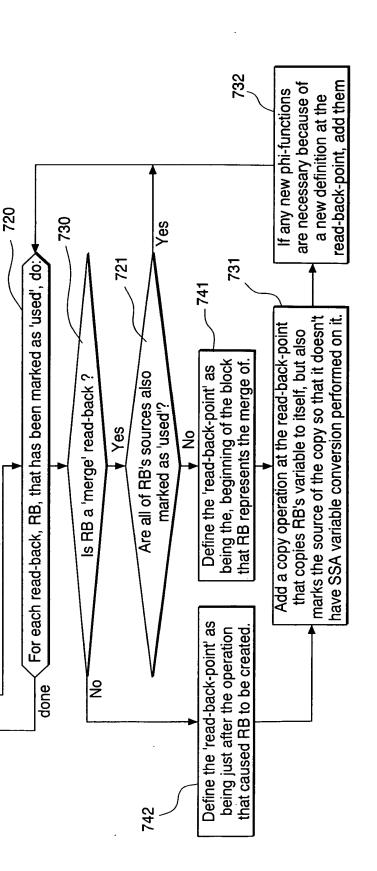


Figure 7A

810 80 822 821 Look up BLOCK in BLOCK_BEGIN_READ_BACKS and BLOCK_END_READ_BACKS, assigning the results to OLD_END_BEGIN_READ_BACKS respectively, and predecessor blocks. Where two or more different read-backs for the same variable are present, a 'merge read-back' is created to 820 Define NEW BEGIN READ BACKS to be the intersection of the BLOCK END READ BACKS value of all of BLOCK's Inputs: BLOCK, ACTIVE_VARS, ALL_ACTIVE_VARS BACKS, initialized from NEW BEGIN READ BACKS for BLOCK in BLOCK BEGIN READ BACKS Make NEW BEGIN READ BACKS the entry using an empty set where BLOCK has no entry Define a new read-back set, NEW END READ combine them, staring at BLOCK BEGIN READ BACKS, or is this the first time BLOCK has been READ BACKS different from OLD Is NEW BEGIN Yes processed? (ઇ The procedure 'propagate_read_backs' ဍ Done 870 871 for BLOCK in BLOCK END READ BACKS Make NEW END READ BACKS the entry Add SUCC to PENDING BLOCKS For each successor, SUCC, in BLOCK's successor list, do: END READ BACKS and OLD END BACKS different? Are NEW \varnothing g 880

Figure 7B
The procedure 'propagate_read_backs'

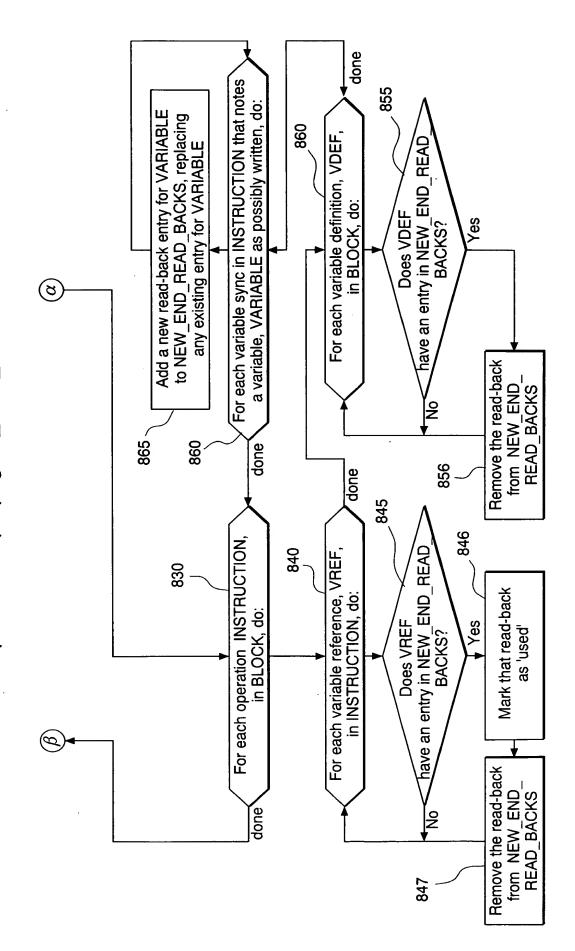


Figure 8 Example source program

This short C program is used to illustrate the invention:

```
extern int g () , h ( ) , i ( ) , x;
  int foo (int *p)
     (*p) ++;
if (*P > 10)
                                                                                    [810]
          g();
          h();
          if (x > 5)
            g();
          if (x > 3)
            i();
          else
            X = *p;
          *P = 5;
     return *p;
  }
Here's the same program converted to a slightly more primitive form:
  int foo (int *p)
  block1:
     p := p + 1;
                                                                                   [B20]
     if (*P <= 10)
```

```
goto block8;
block2:
  g();
  ĥ();
  if (x <= 5)
    goto block4;
block3:
  g();
block4:
if (x > 3)
  goto block6;
block5:
  x := *p;
                                                                            [B40]
  goto block7;
block6:
  i ();
block7:
  p := 5;
                                                                            [830]
block8:
  return *p;
```

Figure 9

SSA converted program, with simple implementation of read-backs:

The following is psuedo-C, augmented with the `phi' operation, where

```
RESULT = phi (block1: VAL1, ..., blockN:VALN)
```

means assign VAL1 to RESULT if control-flow comes from block1', and similarly so on for each value of N.

The extra variables 'pvN", where N is an integer, are SSA versions of *P, and are in fact local variables, not dereferences of p.

```
1nt foo (int *p)
  int pvl, pv2, pv3, pv4, pv5, pv6;
block1;
  pvl = *P + 1;
  if (pvl \ll 10)
    goto block8;
block2:
  *P = pvI:
                   /* This writes-back PV1 to *P. */
  g ();
  pv2 = *P:
                     /* This reads-back *P into PV2. */
  P = pv2;
                     /* This writes-back PV2 to *P. */
  h();
  pv3 = *P:
                     /* This reads-back *P into PV3 */
                                                                              [912]
  if (x \le 5)
    goto block4;
block3:
  p = pv3;
                    /* This writes-back PV4 to *p, */
  g();
                     /* This reads-back *p into PV4. */
                                                                              [911]
  pv4=*p;
block4:
  pv5 = phi (block3: pv4, block2: pv3)
                                                                              [910]
  if (x > 3)
    goto block6;
block5:
  gota block7;
block6;
  i();
block7:
  x = phi (block6: x, block5: pv5);
block8:
  pv6 = phi (block1: pv1, block7: 5);
                    /* This writes-back PV6 to *P. */
  *P = pv6;
  return pv6;
}
```

Figure 10

SSA converted program, with the implementation of read-backs described in this patent

```
int foo (int *p)
  int pv1, pv2, pv3;
block1:
  pv1 = *p +1;
                                                                           [1011]
  if (pvl \ll 10)
    goto block8;
block2;
                     /* This writes-back pv1 to *P. */
  p = pv1;
  g();
                                                                           [1021]
  h();
                                                                           [1022]
  if (x <= 5)
    goto block4;
block3;
                                                                           [1023]
  g ();
block4:
                     /* This reads-back *p into pv2, */
  pv2 = *p;
                                                                           [1030]
  if (x > 3)
    goto block6;
block5;
  goto block7;
block6:
  i();
                                                                           [1024]
black7:
  x = phi (block6 : x, block5 : pv2);
                                                                           [1031]
block8:
                                                                           [1010]
  pv3 = phi (block1: pv1, block7: 5);
  P = pv3;
                     /*This writes-back PV3 to *P */
  return pv3;
```

Figure 11 Register-alloced and SSA-unconverted program

using BBA-form requires having a good register allocator that will merge variables where possible, as it tends to generate a lot of variables with short lifetimes. We assume that here.

```
int foo (int *p)
  int pv;
block1;
  pv = *p + 1;
  if (pv <= 10)
    goto block8;
block2:
                    /* This writes-back pv to *P. */
  'P = pv;
  g();
  h();
  if (x \le 5)
    goto block4;
block3:
  g();
block4:
  if (x > 3)
    goto block6;
block5:
    goto block7;
block6:
  i ();
block7:
   pv = 5;
block8:
  *P= pv
                   /* This writes-back PV to *P. */
return pv;
```

Figure 12
Original SSA-conversion process

